



freeBSD<sup>®</sup> **JOURNAL**  
Nov/Dec 2018



**Introducing  
FreeBSD 12.0**

**The Daemon's Dozen**

**ZPool Checkpoint**

**Debugging the Kernel**



# Table of Contents



Nov/Dec 2018

- 3 Foundation Letter** Welcome to the last *FreeBSD Journal* issue of 2018, and the launch of FreeBSD 12.0.  
*By George Neville-Neil*
- 22 We Get Letters** Dear ed(1), I keep hearing about FreeBSD conferences. Should I go?  
*By Michael W Lucas*
- 24 Conference Report** At EuroBSDcon 2018, BSD developers and users presented a wide range of innovative and original talks—many on topics not previously presented at other European venues. *By Roller Angel*
- 28 New Faces of FreeBSD.** In this issue, we shine a spotlight on Sergey Kozlov who received his ports bit in September and Vinícius Zavam who received his ports bit in October. *By Dru Lavigne*
- 31 Events Calendar**  
*By Dru Lavigne*
- 32 svn Update** For a while there was not much action in HEAD, as the release engineering team had frozen the tree to create a branch of what will become 12-STABLE. That freeze has been lifted and we once again are seeing lots of new features and exciting changes.  
*By Steven Kreuzer*
- The Daemon's Dozen**  
**Introducing FreeBSD 12.0.....4**  
A look at some new features, removed features, and forthcoming changes, starting with storage.  
*By Michael W Lucas*
- Debugging the FreeBSD Kernel.....6**  
It is useful to have some familiarity with FreeBSD kernel debugging even if you are not a FreeBSD developer.  
*By Mark Johnston*
- ZPool Checkpoint.....18**  
In March 2018, Alexander Motin ported the Pool Checkpoint feature of OpenZFS from Illumos to FreeBSD. This article provides a high-level description of what happens under the hood during each administrative operation. *By Serapheim Dimitropoulos*
- FreeBSD 12.0 Toolchain Update.....20**  
FreeBSD 12.0 continues the trend in recent FreeBSD releases of transitioning away from obsolete GPLv2-licensed toolchain components to modern ones.  
*By John Baldwin and Ed Maste*

# Support FreeBSD<sup>®</sup>



## Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

Your investment will help:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Making a donation is quick and easy.  
[freebsd.foundation.org/donate](https://freebsd.foundation.org/donate)



# FreeBSD<sup>®</sup> JOURNAL

## Editorial Board

- John Baldwin • FreeBSD Developer, Member of the FreeBSD Core Team and Co-Chair of *FreeBSD Journal* Editorial Board.
- Brooks Davis • Senior Computer Scientist at SRI International, Visiting Industrial Fellow at University of Cambridge, and member of the FreeBSD Core Team.
- Bryan Drewery • Senior Software Engineer at EMC Isilon, member of FreeBSD Portmgr Team, and FreeBSD Committer.
- Justin Gibbs • Founder of the FreeBSD Foundation, Director of the FreeBSD Foundation Board, and a Software Engineer at Facebook.
- Daichi Goto • Director at BSD Consulting Inc. (Tokyo).
- Joseph Kong • Senior Software Engineer at Dell EMC and author of *FreeBSD Device Drivers*.
- Steven Kreuzer • Member of the FreeBSD Ports Team.
- Dru Lavigne • Director of Storage Engineering at iXsystems, author of *BSD Hacks* and *The Best of FreeBSD Basics*.
- Michael W. Lucas • Author of *Absolute FreeBSD*.
- Ed Maste • Director of Project Development, FreeBSD Foundation.
- Kirk McKusick • Treasurer of the FreeBSD Foundation Board, and lead author of *The Design and Implementation* book series.
- George V. Neville-Neil • President of the FreeBSD Foundation Board, and co-author of *The Design and Implementation of the FreeBSD Operating System*.
- Philip Paeps • Secretary of the FreeBSD Foundation Board, FreeBSD Committer, and Independent Consultant.
- Hiroki Sato • Director of the FreeBSD Foundation Board, Chair of Asia BSDCon, member of the FreeBSD Core Team, and Assistant Professor at Tokyo Institute of Technology.
- Benedict Reuschling • Vice President of the FreeBSD Foundation Board, a FreeBSD Documentation Committer, and member of the FreeBSD Core Team.
- Robert N. M. Watson • Director of the FreeBSD Foundation Board, Founder of the TrustedBSD Project, and University Senior Lecturer at the University of Cambridge.

S&W PUBLISHING LLC  
PO BOX 408, BELFAST, MAINE 04915

- Publisher** • Walter Andrzejewski  
walter@freebsdjournal.com
- Editor-at-Large** • James Maurer  
jmaurer@freebsdjournal.com
- Copy Editor** • Annaliese Jakimides
- Art Director** • Dianne M. Kischitz  
dianne@freebsdjournal.com
- Advertising Sales** • Walter Andrzejewski  
walter@freebsdjournal.com  
Call 888/290-9469

*FreeBSD Journal* (ISBN: 978-0-615-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).

Published by the FreeBSD Foundation,  
5757 Central Ave., Suite 201, Boulder, CO 80301  
ph: 720/207-5142 • fax: 720/222-2350  
email: info@freebsd.foundation.org

Copyright © 2018 by FreeBSD Foundation. All rights reserved. This magazine may not be reproduced in whole or in part without written permission from the publisher.

## LETTER from the Board

Welcome to the last **FreeBSD Journal** issue of 2018 and the launch of **FreeBSD 12.0**. But first, we've got a big announcement. Beginning with the January/February 2019 issue, the **FreeBSD Journal** will be free to everyone. Yes, **Free**.

- Why? you might ask. A big part of the FreeBSD Foundation's mission is to raise awareness of FreeBSD throughout the world. We know that the *FreeBSD Journal* is the best venue for keeping those interested in FreeBSD up-to-date. Our articles are timely and informative. Therefore, the Foundation has decided that the benefit of making it accessible to everyone at no charge will outweigh the cost of producing it. However, donations will be accepted if you'd like to help support the *Journal*.

- That's not all. We're also debuting an enhanced browser-based edition that will provide a much better experience for our readers.

- As we head into 2019, I'd like to thank the *FreeBSD Journal* authors, editors, and publishing staff for all of the hard work this year. Keep an eye out for more announcements as we get closer to the release of the January/February 2019 issue. Questions? Please don't hesitate to contact us.

See you all in 2019!

George Neville-Neil

**President of the FreeBSD Foundation Board of Directors**

By Michael W **LUCAS**

# The Daemon's Dozen

## Introducing FreeBSD 12.0

We've all been eagerly awaiting FreeBSD 12.0 since about two hours after FreeBSD 11.0 escaped. Here I'll take a few looks at some new features, removed features, and forthcoming changes, starting with storage.

**Z**FS users are probably familiar with the `beadm(8)` package, the boot environment administration program. We've previously covered it in the *FreeBSD Journal*. FreeBSD now includes `bectl(8)`, a boot environment control program. It retains much of the `beadm(8)` syntax, letting you create, activate, list, and destroy boot environments. Additionally, it lets you mount unused boot environments and create jails from those boot environments. I've used jails to troubleshoot problematic boot environments, so this is a welcome addition to the base system.

Many people have used GELI to encrypt disks. Doing hard drive encryption properly is tricky, but GELI handily reduces the risk of exposing confidential data when your laptop gets stolen. GELI was painful to use on systems with many hard drives, though. Each disk needed to be configured separately. With FreeBSD 12.0, GELI can now configure multiple disks simultaneously so long as they use the same encryption key, easing this annoyance for sysadmins who administer such machines. Also, in the GELI realm, the installer can now encrypt disks on hosts booting off of UEFI.

FreeBSD's GEOM system lets us stack storage transformations on top of one another in exactly the way you need for your application, hardware, and environment. If you want a single massive striped and encrypted storage array, you could stripe the disks together and then encrypt them—or you could encrypt each disk separately and then stripe them. GEOM empowers you. The catch, of course, comes when you're trying to figure out what you did with that power. The new `geom -t` command displays the host's GEOM stack.

## geom -t

Geom	Class	Provider
ada0	DISK	ada0
ada0	DEV	
ada0	PART	ada0p1
ada0p1	DEV	
ada0p1	LABEL	gpt/gptboot0
gpt/gptboot0	DEV	
ada0	PART	ada0p2
ada0p2	DEV	
swap	SWAP	
ada0	PART	ada0p3
ada0p3	DEV	
zfs::vdev	ZFS::VDEV	

You can easily see how the providers and consumers get stacked onto each other, what goes where, and the labels applied to everything.

Similarly, if you want to see all of the information about a particular GEOM container, use the new `geom -p` command. Here I tell GEOM to spill everything it knows about `da2p1`.

## geom -p da2p1

Some storage changes are not so obvious. A TRIM request is used to tell an SSD or a memory filesystem that a block is no longer used. FreeBSD 12.0 batches and consolidates those requests, reducing disk I/O. You'll see dozens of such minor performance enhancements throughout the release.

FreeBSD 12.0 has one major new feature for storage, though: a parallel NFS (pNFS) server. Parallel NFS can greatly enhance NFS performance but requires a pNFS-aware client. FreeBSD's NFS client is pNFS-aware. It's a brand-new feature, though, and rather than telling you to deploy it immediately I'd encourage you to thoroughly test it in your environment. pNFS also requires NFSv4, so if you're still using an older version of NFS it's time to look at upgrading.

NFS touches on the network, so let's talk networking next.

The FreeBSD 12.0 kernel enables the virtual networking stack VIMAGE by default, allowing you to use multiple routing tables. You can now have your host use its standard routing table but create a new routing table just for jails running on that host. Also, you can now use PF within a jail, and PF's HFSC traffic shaping can now handle up to 100 Gbs.

Eight-bit 10Mbps network cards are no longer popular enough for FreeBSD to support, and their presence in the kernel creates roadblocks for development of current features. Drivers for purely 10Mbps cards like `lmc`, `ixgb`, and `nxge` have all been

dropped from the current kernel. Many drivers for 10/100 Mbs cards such as `ed`, `ep`, `pcn`, and so on are deprecated and salted for removal in FreeBSD 13.0. Twenty years ago, I was a big fan of NE2000-compatible cards. They were essentially indestructible. The problem is that was twenty years ago. It's time to buy a new network card.

FreeBSD has had a long-running problem with video drivers. Video cards come out more frequently than FreeBSD releases. Users sensibly want to use all the fancy features of their new video cards and would get frustrated when a

two-month-old FreeBSD didn't support those drivers. FreeBSD solved this by removing the advanced video drivers from the base operating system and making them a port instead. Ports are much more easily updated than base system components, meaning users get accelerated video and 3D support and all the other fancy stuff they want. Install the `drm-kmod` package for the latest video drivers and see the FreeBSD Graphics Wiki at <https://wiki.freebsd.org/Graphics> for details on how to use it.

If you need to extract information from `last(1)` and `lastlogin(8)`, these programs now support `libxo`. Try `last --libxo xml` to get a sample of the output and use your organization's standard XML tools to gather the information you need.

There's a bunch of other new features and details. The boot loader now includes Lua rather than Forth, so all you Lua hackers can customize the boot process. Some older, standard device drivers have undergone improvements. While the console could previously display text faster than I can read, the `vt(4)` driver now runs up to four times as fast as in FreeBSD 11.0. Contributed programs like `ntpd(8)` and `mandoc(1)` have been upgraded. The random number generator `random(4)` has seen modernization, and the integrated OpenSSL will get support for several years. The default pager is now `less(1)`, and `newsyslogd(8)` ignores `setuid` and executable log files—as it should.

FreeBSD 12.0 has many smaller enhancements, improvements, and tweaks. Really, the best way to see them all is to install it and see what happens. I've been happy with it. I think you will be as well.



**Michael W Lucas** has been using BSD since the 1980s. His most recent book is the third edition of *Absolute FreeBSD*. Learn more at <https://mwli.io>.

# DeBUGGING

## the FreeBSD Kernel

By Mark Johnston

**As I write this, the long-awaited FreeBSD 12.0 release is within days of shipping. Of course, FreeBSD developers have been meticulously polishing and testing 12.0's kernel for months and we're confident that every last bug has been squashed.**

For the typical FreeBSD user, the subject of kernel debugging is just an intellectual curiosity, right? This is how it should be, at least; FreeBSD's stability is one of its main selling points and contributes a lot to its reputation. The stability of an operating system kernel is especially important since a fault in the kernel will typically bring down the entire system. In truth, however, no matter how many bugs were fixed in the lead-up to 12.0, the harsh reality is that some are still lingering and will emerge later to bite users, often with a stress-inducing kernel panic.

It is useful to have some familiarity with FreeBSD kernel debugging even if you are not a FreeBSD developer. The permissiveness of the BSD license has allowed many companies to build products on top of the FreeBSD source tree, often directly extending the kernel. These extensions will, of course, contain bugs or may expose latent bugs in FreeBSD itself. Moreover, developers working on such a code base generally cannot work directly with the FreeBSD community and thus would need to do some debugging themselves, perhaps before reporting the issue upstream. Another very common scenario is the case where developers do not have direct access to the system exhibiting a problem, so live debugging is not possible. Instead, any data required for debugging must be collected by an administrator or user. Knowing what to collect and how to collect it can greatly speed up the debugging process.

Finally, familiarizing yourself with our debugging tools is important for becoming a proficient kernel developer; as a newcomer you will inevitably make mistakes, some trivial and some not, and debugger can often reveal a problem more quickly than staring at one's own code. Kernel bugs can of course have many different causes, but the high-level approach to resolving them is always the same:

1. Find the root cause of the bug or get as close as possible.
2. Find a way to work around the bug, if both possible and necessary.
3. Fix the bug.

Step 2 might be much more important than step 3 if the bug is affecting a production workload, and step 3 might not be achievable; hardware failures and firmware bugs are common sources of what appear to be kernel bugs, and generally cannot be fixed directly. However, step 1, which generally consists of kernel debugging, is a prerequisite of both steps 2 and 3. The task of debugging an operating system kernel might seem daunting, and, indeed, it poses some unique challenges, but methodologically it is the same as debugging any userspace application.

The FreeBSD kernel is a large body of code by most standards, and one cannot rely on code inspection and debug-by-printf to get very far with anything except trivial bugs. The fact that one must reboot in order to test a new kernel also constrains these kinds of primitive debugging techniques. FreeBSD contains various facilities to do post-mortem debugging and to help find reproducible triggers for bugs without a known cause, at which point debugging tools can be brought to bear on the problem. This article aims to highlight some of these facilities and provide some techniques for using them.



## Triggering Kernel Bugs

Unexpected kernel panics are never pleasant. Almost as bad, though, is trying frantically to replicate someone else's kernel panic and failing. Usually this happens when the bug is timing-dependent, as in a race condition or a use-after-free. There is no silver bullet for these situations, but the FreeBSD kernel provides several facilities that may prove useful.

### INVARIANTS Kernels

**INVARIANTS** is a kernel option that is configured in development builds of the FreeBSD kernel—if you are not running a FreeBSD release, it is quite likely that you are using **GENERIC** with **INVARIANTS** configured. In this case, congratulations! Your kernel has thousands of integrity checks enabled, constantly testing the assumptions of the programmers who wrote it. These checks impose runtime overhead, making **INVARIANTS** unsuitable for many production workloads, but they should be enabled whenever it is feasible. If you report a FreeBSD kernel panic to the mailing lists, it is quite likely that you will simply be asked to test with an **INVARIANTS** kernel if you are not already doing so. Other kernel options, such as **DIAGNOSTIC** and **DEBUG\_VFS\_LOCKS**, enable further rounds of more expensive integrity checking that can catch many bugs.

### Memory Trashing

The FreeBSD kernel has an implementation of `malloc(9)` very similar to that used in regular C applications. Additionally, a memory subsystem called UMA ("Universal Memory Allocator") provides a slab allocation API for high-performance code paths. When **INVARIANTS** is configured in the kernel, memory freed using either of these APIs is "trashed," meaning that its contents is overwritten with a specific pattern. Any time trashed memory is reused, its contents are verified against that pattern. If verification fails, it means that the memory was written to after it was freed, and the kernel panics.

This is a very standard technique and is implemented in most full-featured memory allocators. On FreeBSD, the trash pattern is `0xdeadb0de` so that, while the allocator itself will catch a write-after-free, anything that reads a trashed memory location (i.e., a read-after-free) is also likely to crash if it treats the value as a pointer. When debugging or triaging a kernel panic, keep an eye out for `0xdeadb0de`—it very often signifies a use-after-free.

Memory trashing helps use-after-frees stand out by making it clear what the use-after-free did. Consider the following memory dump:

<code>0xc00000037d3c0ba0:</code>	<code>0</code>	<code>fffffe00017f2530</code>
<code>0xc00000037d3c0bd0:</code>	<code>ffffff80002268e80</code>	<code>fffffffff80b35c58</code>
<code>0xc00000037d3c0be0:</code>	<code>deadb0dedeadb0de</code>	<code>deadb0dedeadb0de</code>
<code>0xc00000037d3c0bf0:</code>	<code>deadb0dedeadb0de</code>	<code>deadb0dedeadb0de</code>
<code>0xc00000037d3c0c00:</code>	<code>0</code>	<code>deadb0dedeadb0de</code>
<code>0xc00000037d3c0c10:</code>	<code>deadb0dedeadb0de</code>	<code>deadb0dedeadb0de</code>
<code>0xc00000037d3c0c20:</code>	<code>deadb0dedeadb0de</code>	<code>deadb0dedeadb0de</code>
<code>0xc00000037d3c0c30:</code>	<code>deadb0dedeadb0de</code>	<code>deadb0dedeadb0de</code>

Here we can see that the write-after-free wrote 8 bytes of zeros at a specific offset within a structure; that can be a helpful clue for identifying the responsible code.

A use-after-free is an especially nasty type of bug since the cause and effect of the bug may be far removed from each other. Memory trashing helps identify that a use-after-free occurred but can't catch the offending code in the act. This is where MemGuard comes in.

### MemGuard

MemGuard is a memory subsystem feature which proactively detects use-after-frees for a particular memory type. A memory type is defined by either the type argument to `malloc(9)`, or a UMA zone. For example, the line:

```
kq = malloc(sizeof *kq, M_KQUEUE, M_WAITOK | M_ZERO);
```

references `M_KQUEUE`, which is defined with:

```
MALLOC_DEFINE(M_KQUEUE, "kqueue", "memory for kqueue system");
```

The `M_KQUEUE` `malloc(9)` type is thus named "kqueue". Statistics for each `malloc(9)` type can be listed with: `vmstat -m`, and similar statistics for UMA zones can be listed with `vmstat -z`.

MemGuard is not present in FreeBSD kernels by default and must be compiled in by adding the

`DEBUG_MEMGUARD` option to the kernel configuration. Then, to configure MemGuard for a particular memory type, set `vm.memguard.desc` to the name of the `malloc(9)` type or UMA zone, either by adding an entry to `/boot/loader.conf` or by setting the `sysctl`. Once MemGuard is configured, it hooks all allocations and frees of the memory type. Each allocation is padded to a full page, and when a piece of memory is freed, the entire range of allocated memory is unmapped. This means that an attempt to access the memory after it has been freed will likely result in a page fault which crashes the kernel, catching the use-after-free as it happens.

The MemGuard facility is quite powerful but is correspondingly heavy-handed. Since each memory allocation must be padded to a full page, it will waste a large amount of memory if the memory type is used for small allocations. For instance, enabling MemGuard on the `mbuf` zone causes  $4096 - 256 = 3840$  bytes to be wasted per `mbuf`. The MemGuard allocation and free hooks will also severely impact the performance of the allocation: unmapping kernel memory is an expensive operation that must be serialized across all CPUs in the system. Thus, on a system with many CPUs and at a high allocation rate, MemGuard can easily become a major bottleneck. If the use-after-free is timing-dependent as is often the case, MemGuard's overhead can be such that the bug stops occurring. Nonetheless, it is quite effective in practice.

## Fail Points

Many bugs, both in the kernel and userspace, are the result of incorrect or insufficient error handling. These bugs lay dormant until the corresponding error occurs, so the commonly executed "happy path" is unaffected. When the error does occur, it may be the result of a hardware or firmware malfunction, or some other seemingly unlikely event. Suppose you are in the position of having to debug the error-handling problem, and against all odds, you think that you have spotted the bug and want to verify your theory. How do you go about this? How can you write a regression test to ensure that your fix remains valid into the foreseeable future?

FreeBSD's fail point subsystem (documented in `fail(9)`) is a useful tool here. Fail points are a mechanism that allows programmers to inject errors and modify behavior at pre-defined locations in the kernel code. Fail points do not have any effect by default, but can be activated using `sysctl(8)`: each fail point is controlled by a unique `sysctl`, usually under `debug.fail_point`. Fail points can be used to modify stack variables (usually to inject an error), change control flow (by following a `goto` or returning early from a function), or sleep or pause (useful for triggering race conditions).

The fail point interface consists of a set of macros, most of which are convenience wrappers for `KFAIL_POINT_CODE`. For example, we might try using a fail point to simulate random failures from `malloc(9)`. The following fail point at the end of `malloc()` would do the trick:

```
void *
malloc(size_t size, struct malloc_type *mtp, int flags)
{
    ...
    if ((flags & M_WAITOK) == 0)
        /* M_WAITOK allocations are not allowed to fail. */
        KFAIL_POINT_CODE(DEBUG_FP, malloc, {
            free(va, mtp);
            va = 0;
        });
    return ((void *)va);
}
```

The same fail point could be defined a bit more succinctly:

```
void *
malloc(size_t size, struct malloc_type *mtp, int flags)
{
    ...
    KFAIL_POINT_CODE_COND(DEBUG_FP, malloc, (flags & M_WAITOK) == 0, 0, {
        free(va, mtp);
        va = 0;
    });
    return ((void *)va);
}
```

Either way, we must take care to free the newly allocated memory ourselves before simulating an allocation failure: otherwise that memory would be permanently leaked. For ad-hoc debugging, this might be acceptable, but in general, one must take care to ensure that a fail point does not introduce a new bug.

With this fail point defined, we get a `sysctl` named `debug.fail_point.malloc`. Fail points are activated by setting the corresponding `sysctl` to a string which describes the action to take when a thread executes the fail point. The default action is "off," meaning that the fail point has no effect. The "return" action causes the fail point's code to be executed; other actions are defined in the manual page. The kernel would not survive for long if all `malloc(M_NOWAIT)` allocations failed, so fail points can be configured to execute with a specified probability. The setting

```
# sysctl debug.fail_point.malloc='1%*return'
```

would cause the fail point's code to be executed roughly once for every 100 `malloc(M_NOWAIT)` calls.

Fail points are particularly useful for simulating disk errors; such errors are hopefully quite rare, but filesystems, among other things, must handle them robustly. When properly integrated, fail points can be an important part of the regression testing regimen for such code. For example, FreeBSD's RAID1 implementation `gmirror` contains fail points used to simulate errors from the underlying disks. Over time, a number of bugs in `gmirror`'s error handling have been fixed, and fail points are now used in several automated `{gmirror}` tests.

## Kernel Stack Tracing

Occasionally a kernel bug will manifest with a thread entering an infinite loop in the kernel. No crash occurs, but a keen observer might report the issue as a task in `top(1)` consuming 100% of a CPU for no obvious reason. If the system is not being used in production, it might be reasonable to drop in to a kernel debugger to interrogate the task in question, but this is not always possible.

FreeBSD has the ability to print kernel stacks without pausing the system using the `procstat(1)` utility's `-k` option. `procstat -kk <pid>` prints kernel stacks, and `procstat -kka` prints kernel stacks for every thread in the system. While this does not directly help in finding a trigger for the bug, it provides an accessible starting point both for tasks that appear "stuck" and are unresponsive, and for tasks that appear to be spinning forever. When `procstat(1)` is asked to capture the stack for a given thread and the thread is currently running on a CPU, `procstat(1)` will raise an interrupt on that CPU and capture the thread's stack frames from the interrupt handler. While this may not provide sufficient information to root-cause a bug on the spot, it is often a useful clue, and the fact that it can be used without bringing down the entire system makes it a handy place to begin an investigation.

## Kernel Dumps and kgdb

`kgdb` is the workhorse of many kernel debugging sessions, especially when live debugging is impossible. As the name suggests, `kgdb` is an extension of `gdb`, the GNU debugger, for FreeBSD kernels. It is actively maintained and provided as a component of the `gdb` package. Similar to `gdb`, it can be used to inspect the running kernel or to inspect the post-mortem output of a kernel panic, usually referred to as a "kernel dump" or "crash dump." If you hit a kernel panic, you may be requested to provide a kernel dump to a developer for analysis; the logistics of doing so can be complicated, so it is worth understanding the various options available for saving kernel dumps.

Kernel dumps are effectively a complete copy of the kernel's state at the time of a panic and often contain enough information to completely piece together the cause of the crash. When a kernel panic occurs, the kernel will log a message and, if so configured, jump to a special routine which attempts to save a dump and then automatically reboots the system. Upon a subsequent boot-up, `savecore(8)` will check for and recover a previously saved dump, storing it in `/var/crash`.

Kernel dumps come in two flavors: minidumps and full dumps. Full dumps are largely an anachronism and are used on architectures that do not yet have minidump support. The difference is that full dumps contain the entire contents of RAM, while minidumps contain only memory which was allocated to the kernel at the time of the crash: unallocated pages, user memory and cached files are omitted. Minidumps are correspondingly smaller but more complicated to implement and support. `kgdb` can handle both types, so the distinction is not usually important or even noticeable. It's important to note that kernel dumps will generally contain sensitive user data and should be shared cautiously.

The main consideration when configuring kernel dumps is the location at which the dump will be saved by the kernel. Because the code which actually saves the dump is running after a kernel panic, it cannot rely on the kernel to be operational and thus must be as simple as possible; we definitely don't want it to try writing

to a filesystem! The simplest approach is to save the dump on the system's swap partition: the contents of swap are, by definition, useless after a reboot, so there is no harm in it being overwritten. This behavior can be configured by adding the line

```
dumpdev="AUTO"
```

to `/etc/rc.conf` —if a swap device is configured in `/etc/fstab`, it will then be used for kernel dumps. Otherwise, to save dumps to a local disk, the name of the disk device must be specified instead of "AUTO."

When configuring a dump device, it's a good idea to test it by triggering a panic and verifying that a `vmcore` file appears in `/var/crash` after the reboot:

```
# sysctl debug.kdb.panic=1
```

The main difficulty with a local dump device is storage space. The size of a kernel dump is roughly proportional to the amount of RAM in the system. While minidumps greatly reduce the need for space, they can still easily be in the tens of gigabytes on a system with 128GB of RAM, larger than a reasonably sized swap partition. A system administrator would be forced to either waste disk space or potentially lose the ability to save kernel dumps. Fortunately, FreeBSD 12.0 comes with several new features to help alleviate this problem. First, the code which saves a kernel dump can now compress the data before writing it, greatly reducing space requirements. Currently, the `zlib` and newfangled `Zstandard` algorithms are supported for this purpose. `Zstandard` is both faster and better at compressing kernel dumps, and we frequently see compression ratios of 8:1 or better. Either algorithm can be configured by setting `dumpon_flags` in `/etc/rc.d`; see the `dumpon(8)` manual page for details. `Zstandard` compression can thus be easily configured:

```
dumpdev="AUTO"  
dumpon_flags="-Z"
```

Compression staves off the storage requirement problem for now, but the fundamental problem remains. Moreover, FreeBSD is used on many systems that do not have a suitable dump device to begin with. For example, many embedded devices—such as routers—boot from read-only media and do not have any dynamic storage. Virtual machines are often provisioned without swap space, and diskless servers boot from an NFS mount and do not have any local storage at all. To address these scenarios, FreeBSD 12.0 will come with the ability to transmit kernel dumps over a network interface using an IPv4 UDP-based protocol called `Netdump`.

## Netdump

`Netdump` provides flexibility to system administrators and developers by allowing them to use a remote server to receive kernel dumps from clients (panicking kernels). `Netdump` is integrated with the existing kernel dump facilities and is transparent with respect to the kernel dump features described earlier; it is possible to use kernel dump compression or encryption with `Netdump`, just as with dumps to a local disk. From an administrator's perspective, `Netdump` simply requires additional configuration.

`Netdump` clients use `dumpon(8)` to configure client parameters. Instead of specifying a block device, specify the name of the network interface to be used. This should preferably be a low-traffic management interface rather than one of the system's data ports: `Netdump` will be less reliable if the interface that it uses was busy at the time of the panic. `Netdump` does not automatically support every network adapter, but many commonly encountered adapters are, indeed, supported, and a list of supported drivers can be found in the `netdump(4)` manual page. Of course, the network interface name is not sufficient: the `Netdump` client code needs to know how to contact the server! Three network address parameters must be specified:

- The client IP address to use (`-c`). This parameter is mandatory.
- The server's hostname or IP address (`-s`). This parameter is mandatory.
- The IP address of the first-hop router between the client and server (`-g`). That is, the address of the router that forwards packets from the client to the server. This is typically just the system's default router as listed in the output of `netstat -rn`. This parameter is optional: if it is not specified, `dumpon(8)` uses the system's default route if one is configured, and, otherwise, assumes that the client and server are on the same link.

These parameters can all be specified in `/etc/rc.conf`:

```
dumpdev="em0"
dumpon_flags="-s 192.168.1.34 -c 192.168.2.154 -g 192.168.2.1"
```

At the time of writing, only IPv4 parameters are supported. We hope to support IPv6 in future releases of FreeBSD.

Because the Netdump client code runs only after the system has panicked, it cannot use any of the kernel code which normally does the work of building IPv4 packets and sending them out onto the wire. In particular, it cannot make use of the kernel's ARP cache, so as a first step, it must resolve the Ethernet address of the next hop. In fact, the Netdump client always attempts to resolve the server directly before trying to resolve the gateway. This way, if both the client and server happen to be on the same link, they can communicate directly. Otherwise, the client relies on the gateway to forward all Netdump packets to the server.

The Netdump server program is called `netdumpd` and is available as a package for FreeBSD 11 and 12. It can be configured via `/etc/rc.conf`; its main parameter specifies the root directory under which it will save all kernel dumps from clients. `netdumpd` listens on UDP port 20023 for **HERALD** messages from a client (described below). Using an ephemeral port, it replies to all client messages with an **ACK** message. When a client receives the **ACK** for its initial **HERALD**, it sends all subsequent messages to the UDP port from which the **ACK** originated. When a Netdump session completes successfully, the server saves two files under its root directory: a `vmcore` file, identical to those created by `savecore(8)`, and a text file (`info`) containing metadata describing the kernel dump. It may optionally be configured to execute a user-defined script upon completion, successful or otherwise. This can be used, for example, to send an email to the owner of a system when that system panics and transmits a kernel dump.

The Netdump protocol is extremely simple by design and closely resembles the TFTP protocol. All messages consist of a fixed-size header. Client messages begin with:

```
struct netdump_msg_hdr {
    uint32_t  mh_type;
    uint32_t  mh_seqno;
    uint64_t  mh_offset;
    uint32_t  mh_len;
    uint32_t  mh__pad;
};
```

while server messages are always 4 bytes in length:

```
struct netdump_ack {
    uint32_t  na_seqno;
};
```

There are five Netdump client message types:

- A client sends a **HERALD** message to signal the beginning of a kernel dump. It contains an optional payload: a relative path to the server's root directory under which to save the dump. The path must exist on the server.
- A client sends a **FINISHED** message when it has completed the dump. Upon receipt of this message, the server flushes its output files and invokes the user-specified completion hook.
- **VMCORE** messages contain kernel dump data. The `mh_offset` field specifies the relative offset of the data within the output file.
- The **KDH** ("kernel dump header") message contains kernel dump metadata such as the client's version and panic message.
- The **EKCD\_KEY** contains the encrypted kernel dump key if kernel dump encryption was configured on the client. This message is not used otherwise.

Server messages are very simple: they acknowledge receipt of a client message by including the corresponding sequence number.

## kgdb

After going through all the work to trigger a panic and obtain a kernel dump, it's time to do some actual debugging. With **kgdb**, opening the kernel dump is just like opening a userland core dump using **gdb**: you simply point **kgdb** at the kernel binary and **vmcore** files:

```
# kgdb /boot/kernel/kernel /var/crash/vmcore.0
...
(kgdb) bt
#0  __curthread ()
#1  doadump (textdump=16777216)
#2  0xffffffff804ccdbc in db_fncall_generic (...)
#3  db_fncall (...)
#4  0xffffffff804cc8f9 in db_command (...)
#5  0xffffffff804cc674 in db_command_loop ()
#6  0xffffffff804cf91f in db_trap (...)
#7  0xffffffff8088b553 in kdb_trap (type=9, code=0, tf=<optimized out>)
#8  0xffffffff80badce1 in trap_fatal (frame=0xfffffe00188da6b0, eva=0)
#9  0xffffffff80bad1dd in trap (frame=0xfffffe00188da6b0)
#10 <signal handler called>
#11 0xffffffff808594cf in callout_process (now=1960148907297521)
#12 0xffffffff80c329b8 in handleevents (now=1960148907297521, fake=0)
#13 0xffffffff80c33231 in timercb (et=<optimized out>, arg=<optimized out>)
#14 0xffffffff80bbb289 in hpet_intr_single (arg=0xfffffe0000b590b0)
#15 0xffffffff80bbb32e in hpet_intr (arg=0xfffffe0000b59000)
#16 0xffffffff8080953d in intr_event_handle
    (ie=0xfffff80002169100, frame=0xfffffe00188da9a0)
#17 0xffffffff80c6b748 in intr_execute_handlers
    (isrc=0xfffff80002034e48, frame=0xfffffe00188da9a0)
#18 0xffffffff80c71d54 in lapic_handle_intr (vector=<optimized out>,
    frame=0x6f6bec1fb41fa)
#19 <signal handler called>
#20 acpi_cpu_c1 ()
#21 0xffffffff804f28a7 in acpi_cpu_idle (sbt=<optimized out>)
#22 0xffffffff80c684af in cpu_idle_acpi (sbt=59992010)
#23 0xffffffff80c68567 in cpu_idle (busy=0)
#24 0xffffffff80873c65 in sched_idletd (dummy=<optimized out>)
#25 0xffffffff808069a3 in fork_exit (...)
```

As with **gdb**, debug info is needed in order for **kgdb** to make sense of a kernel dump's contents. In recent versions of FreeBSD, these symbols are stored externally, in `/usr/lib/debug/boot/kernel`, assuming that the kernel being debugged is in `/boot/kernel`. When sending a kernel dump to a developer, it is important to include the contents of both `/boot/kernel` and `/usr/lib/debug/boot/kernel`.

**kgdb** is smart enough to unwind through trap frames, printed above as `<signal handler called>`. These occur when the CPU interrupted the running thread, saving its state on the stack. In the example, we have two such frames: first, a timer interrupt interrupted an idle CPU, and, while processing pending callouts, that CPU took a protection fault caused by a corrupted pointer:

```
(kgdb) frame 11
#11 0xffffffff808594cf in callout_process (now=1960148907297521)
510                                     LIST_REMOVE(tmp, c_links.le);
(kgdb) p tmp
$1 = (struct callout *) 0x11777be9162acbc1
(kgdb) x tmp
0x11777be9162acbc1:      Cannot access memory at address 0x11777be9162acbc1
```

Most of the usual **gdb** commands work in **kgdb**. For instance, `info threads` will enumerate all of the system's threads, including kernel-only threads, and one can switch between threads. As an extension,

`kgdb` can identify threads based on FreeBSD thread ID. This is handy when a data structure points to a particular thread and you want to switch to that thread; for example, when the current thread is blocked on a lock, we may wish to look at the thread holding that lock:

```
(kgdb) p kq
$5 = (struct kqueue *) 0xffffffff801cb5a4a00
(kgdb) p/x kq->kq_lock
$6 = {
    lock_object = {
        lo_name = 0xffffffff814e8f66,
        lo_flags = 0x1430000,
        lo_data = 0x0,
        lo_witness = 0x0
    },
    mtx_lock = 0xffffffff801e3b34000
}
(kgdb) p ((struct thread *)kq->kq_lock.mtx_lock)->td_tid
$7 = 101322
(kgdb) tid 101322 # Switch to the thread with TID 101322.
(kgdb)
```

`kgdb` offers scripting facilities: GDB script, which is the native GDB command language with some standard programming constructs like if statements and while loops, and a Python API. The GDB scripting language is rather primitive and limited, so we will focus on the more powerful Python integration and provide several examples. The API documentation is quite thorough, so the examples merely portray some possible uses for the API without explaining each detail of how they work.

The kernel's `struct thread` contains many fields specific to one subsystem; kernel developers effectively allocate thread-local storage by adding fields to this rather large structure. When debugging, a common need is to find the pointer to the currently selected thread. This can often be done in an ad-hoc way, by searching various frames of the stack for a copy of the pointer, but this is tedious and not scriptable. Instead, let's use Python to add a "convenience function" to behave like the magic `curthread` variable in the kernel:

```
import gdb

def _queue_foreach(head, field, headf, nextf):
    elm = head[headf]
    while elm != 0:
        yield elm
        elm = elm[field][nextf]
def list_foreach(head, field):
    return _queue_foreach(head, field, "lh_first", "le_next")
def tailq_foreach(head, field):
    return _queue_foreach(head, field, "tqh_first", "tqe_next")

def tdfind(tid, pid=-1):
    td = tdfind.cached_threads.get(int(tid))
    if td is not None:
        return td

    allproc = gdb.lookup_global_symbol("allproc").value()
    for p in list_foreach(allproc, "p_list"):
        if pid != -1 and pid != p['p_pid']:
            continue
        for td in tailq_foreach(p['p_threads'], "td_plist"):
            ntid = td['td_tid']
            tdfind.cached_threads[int(ntid)] = td
            if ntid == tid:
```

```

        return td
tdfind.cached_threads = dict()

class curthread(gdb.Function):
    def __init__(self):
        super(curthread, self).__init__("curthread")
    def invoke(self):
        return tdfind(gdb.selected_thread().ptid[2])
curthread() # Register the function with kgdb.

```

The API provides a way to get the selected thread's thread ID, but we must traverse the kernel's data structures ourselves to get the corresponding `struct thread` pointer. This is done by the `tdfind()` routine, which emulates a kernel function of the same name to search the kernel's global list of processes (`allproc`) and find the thread with the specified thread ID. The global list can be quite large, so we cache all threads encountered during the search to speed up future lookups.

The `curthread` class provides the requisite glue: when `$curthread()` is invoked it simply returns the result of calling `tdfind()` on the selected thread's thread ID:

```

(kgdb) p $curthread()
$1 = (struct thread *) 0xffffffff801e3b34000

```

Now we need not grovel through the stack to find the current thread pointer, and the lookup code can be reused for other purposes.

With the addition of `VIMAGE` to the standard kernel configuration, many global data structures in the network are virtualized: every reference to such a global structure is actually relative to the current `vnet`, so there is some implicit context to each reference. The kernel uses a set of macros in `vnet.h` to handle this, but `kgdb` doesn't know about them; the upshot of all this is that the following doesn't work:

```

502         if (pcbinfo == &V_tcbinfo) {
503             INP_INFO_RLOCK_ASSERT(pcbinfo);
(kgdb) p V_tcbinfo
No symbol "V_tcbinfo" in current context.

```

We can get close, though, by providing a convenience function to resolve symbols in the context of the current `vnet`, or in a user-specified `vnet` if that happens to be useful:

```

class vimage(gdb.Function):
    def __init__(self):
        super(vimage, self).__init__("V")

    def invoke(self, sym, vnet=None):
        sym = sym.string()
        if sym.startswith("V_"):
            sym = sym[len("V_"):]
        if gdb.lookup_symbol("sysctl__kern_features_vimage")[0] is None:
            return gdb.lookup_global_symbol(sym).value()

        if vnet is None:
            vnet = tdfind(gdb.selected_thread().ptid[2])['td_vnet']
        if not vnet:
            # If curthread->td_vnet == NULL, vnet0 is the current vnet.
            vnet = gdb.lookup_global_symbol("vnet0").value()
        base = vnet['vnet_data_base']
        entry = gdb.lookup_global_symbol("vnet_entry_" + sym).value()
        entry_addr = entry.address.cast(gdb.lookup_type("uintptr_t"))
        ptr = gdb.Value(base + entry_addr).cast(entry.type.pointer())
        return ptr.dereference()
vimage() # Register the function with kgdb.

```

This snippet effectively reimplements what the real **VIMAGE** implementation does: get a pointer to the current **vnet** from the current thread and do a bit of math to find that **vnet**'s copy of the desired structure. It also tries to provide sane behavior if the kernel does not support **VIMAGE** by simply returning the global structure. Additionally, the desired **vnet** may be explicitly specified if needed. While we cannot simply print **V\_tcbinfo**, we can now get pretty close:

```
(kgdb) p $V("tcbinfo")
$1 = {
  ipi_lock = {
    lock_object = {
      lo_name = 0xffffffff8125b55e "tcp",
      ...
    }
  }
}
```

As a final example, we will implement the **acttrace** command from DDB, which prints the backtraces from all threads that were actively executing on a CPU at the time of the panic. This is often a very important piece of the picture, as many bugs are the result of errant interactions between multiple threads. With luck, the threads that caused the panic will have been on-CPU at the time of the panic.

This example adds a new top-level command, mimicking DDB's interface. It uses the per-CPU (**pcpu**) structures which store information about the execution state of each CPU in the system.

```
import math

def cpu_foreach():
    all_cpus = gdb.lookup_global_symbol("all_cpus").value()
    bitsz = gdb.lookup_type("long").sizeof * 8
    maxid = gdb.lookup_global_symbol("mp_maxid").value()

    cpu = 0
    while cpu <= maxid:
        upper = cpu >> int(math.log(bitsz, 2))
        lower = 1 << (cpu & (bitsz - 1))
        if (all_cpus['__bits'][upper] & lower) != 0:
            yield cpu
        cpu = cpu + 1

class acttrace(gdb.Command):
    def __init__(self):
        super(acttrace, self).__init__("acttrace", gdb.COMMAND_USER)

    def _inferior_thread_by_tid(self, tid):
        threads = gdb.inferiors()[0].threads()
        for td in threads:
            if td.ptid[2] == tid:
                return td

    def invoke(self, arg, from_tty):
        # Save the currently selected thread.
        curthread = gdb.selected_thread()

        # Note: not all platforms have a __pcpu array.
        pcpu = gdb.lookup_global_symbol("__pcpu").value()
        for cpu in cpu_foreach():
            td = pcpu[cpu]['pc_curthread']
            p = td['td_proc']
            self._inferior_thread_by_tid(td['td_tid']).switch()
```

*Continues next page*

Continued

```
        print("Tracing command {} pid {} tid {} (CPU {})".format(
            p['p_comm'].string(), p['p_pid'], td['td_tid'], cpu))
    gdb.execute("bt")
    print

    # Switch back to the starting thread.
    curthread.switch()
acttrace() # Register the command with kgdb.
```

The idea is to iterate over all online CPUs in the system, looking up the per-CPU structure and using that to find the thread that is on the CPU. We use the thread's ID to look up `kgdb`'s representation of that thread, switch to it, and print the backtrace. Once this is all done, we switch back to the starting thread to avoid confusing the user.

```
(kgdb) acttrace
Tracing command idle pid 10 tid 100002 (CPU 0)
#0  cpustop_handler ()
#1  0xffffffff80928074 in ipi_nmi_handler ()
#2  0xffffffff808c0e19 in trap (frame=0xffffffff8104b160 <nmi0_stack+3888>)
#3  <signal handler called>
#4  acpi_cpu_idle_mwait (mwait_hint=0)
...
```

## Conclusion

Hopefully this article has helped illuminate several stages of the typical FreeBSD kernel developer's debugging workflow. Of course, experience and intuition can get you pretty far, but good tooling is tremendously helpful in tracking down the more slippery bugs that arise, especially when they lie outside one's domain of expertise. Debuggers also help attune one to various patterns and details that underlie the core functionality of the kernel, so with experience it becomes easier to spot things that "just don't look right": clues which might lead to a root cause. *Happy debugging!* •

**Mark Johnston** is a freelance FreeBSD developer living in Toronto, Canada. In his spare time he enjoys attempting to make nice sounds on a cello, running, travelling, and playing dodgeball on a team with his friends.

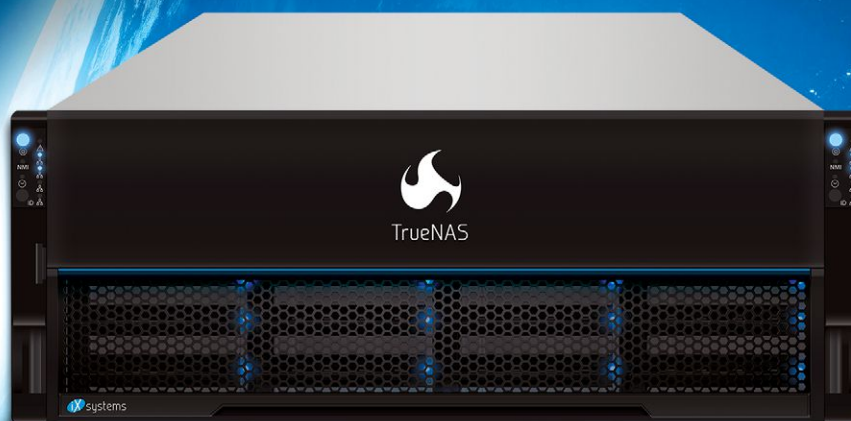


# HAPPY NEW YEAR

Groundbreaking TrueNAS® M Series



# DISRUPTING STORAGE AT WARP SPEED



## LOWEST TCO SHARED STORAGE

Intel® Xeon® Scalable Family Processors

### RESILIENT

- Self-healing Data
- Continuous Operation
- Easy Replication

### WARP FACTOR IX

- Faster than SSD-based Hybrid Storage Arrays
- Flash-Turbocharged Data Access

### EXPANDABLE

- Scales to 10PB using HGST Drives
- 10Gb/s-100Gb/s per NAS
- Non-disruptive Upgrades

### COMPATIBLE

- Citrix, Veeam, & VMware Certified
- Unifies File, Block, & S3 Data
- Supports Leading Cloud Providers

Visit [ixsystems.com/TrueNAS](https://ixsystems.com/TrueNAS) or call (855) GREP-4-iX today!



# zpool CHECKPOINT

By Serapheim **Dimitropoulos**

In March of this year (2018), Alexander Motin ([mav@freebsd.org](mailto:mav@freebsd.org)) ported the Pool Checkpoint feature of OpenZFS from Illumos to FreeBSD. A pool checkpoint can be thought of as a “pool-wide snapshot” that captures the entire state of the pool at the time the checkpoint is created, allowing the user to revert the entire pool back to that state or discard it. The generic use cases are administrative tasks, like OS upgrades, that involve actions that change or destroy ZFS state and metadata. Examples of such actions are: enabling new pool features, changing properties of datasets, or destroying snapshots and filesystems. Before undertaking such actions, administrators can create a checkpoint of their pool and then apply their changes. If something goes wrong with the upgrade, the administrator can then rewind back to the checkpoint as if the actions had never been taken. In the same way a snapshot can help you return user data to a previous state, the checkpoint can help you return the ZFS pool to a previous state.

There are already two tutorials online demonstrating how to use this feature, and a block comment in the source code that gives a high-level overview of its implementation (see the References box). This article lies somewhere in the middle, giving a high-level description of what happens under the hood during each administrative operation.

## Checkpointing a Pool

In ZFS we keep track of changes over time with Transaction Groups (aka TXGs). During each TXG, ZFS accumulates changes in memory and, when certain conditions are met, it syncs those changes to disk, then opens the next TXG. Each data block

records the TXG during which it was created, called its birth TXG. Birth TXGs are important for the pool checkpoint feature because they help us differentiate between blocks created before and after a checkpoint. Finally, a TXG’s last step when syncing to disk is writing the new uberblock to the beginning and end of each disk in an area called the ZFS label, which is different from the EFI label. Each uberblock is the root of the tree of the entire state of the pool for that TXG. Uberblocks are used during pool import as a starting point to find the latest version of all of the pool’s data and metadata.

Whenever an administrator checkpoints a pool (with the “**zpool checkpoint**” command), ZFS copies the uberblock of the current TXG to an area within the pool’s state called the Meta-Object Set (MOS). When this change is synced to disk, the uberblocks from subsequent TXGs reference the “checkpointed” uberblock through the MOS.

## The Lifetime of a Block in a Checkpointed Pool

When a checkpoint exists, the process of allocating new blocks stays the same, but the process of freeing blocks is different. We can’t free blocks that belong to the checkpoint, because we want to be able to rewind back to that point in time. At the same time, we can’t stop freeing blocks entirely, because that would fill up the pool. Thus, every time we are about to free a block, we look at its birth TXG and compare it to the TXG of the “checkpointed” uberblock that we saved in the MOS. If the block was born at or before the TXG of the checkpointed uberblock, it means that the block is part of the checkpoint (i.e., it is referenced

by the checkpointed uberblock). These “checkpointed” blocks are never actually freed. Instead, we add their ranges to lists on-disk we call checkpoint spacemaps (there is one per `vdev`) and leave their segments marked as allocated so they are not reused by the pool. Blocks whose birth TXGs are after the checkpoint’s TXG, are not part of the checkpointed state and can be freed normally.

## Rewinding to the Checkpoint

If administrators want to rewind back to the checkpoint, all they need to do is to export and then re-import the pool with the rewind option (`zpool import --rewind-to-checkpoint`). In this case, the import process takes place as usual, but with one additional step. ZFS first looks at the current uberblock and from that it finds the checkpointed uberblock in the MOS. Then it uses the checkpointed uberblock instead of the current uberblock for the import process, effectively rewinding the pool back to the checkpointed state. Once the import process is done, the rewind is complete. The checkpoint spacemaps no longer exist as they were created after the checkpointed uberblock (which is now the current uberblock). For the same reason, there is no checkpoint uberblock in the MOS. This means that after rewinding there is no additional cleanup and the pool no longer has a checkpoint.

It is also possible to import a checkpoint read-only to access the pool in the state it existed at the time of the checkpoint without actually undoing all of the changes that have happened since the checkpoint was created. This can allow the administrator to recover specific data or a filesystem that was destroyed without rolling back the entire pool.

## Discarding the Checkpoint

If administrators decide to get rid of the checkpoint, they run the discard command

(`zpool checkpoint --discard`). The command instructs ZFS to get rid of the checkpointed uberblock from the MOS. At that point, the pool is considered to no longer have a checkpoint, which allows blocks to be freed normally regardless of their birth TXG. ZFS will also free all the previously recorded ranges from the checkpoint spacemaps—the blocks that we couldn’t actually free because they were referenced by the checkpointed uberblock. The number of these blocks can be quite large depending on how long the checkpoint has existed and how many changes have been made to the pool. Freeing them all in a single TXG would be expensive. Instead, ZFS spawns a thread that frees them over the course of multiple TXGs by prefetching them into memory in chunks and freeing one chunk per TXG.

## Acknowledgments

The development of this feature would not have been possible without the help of my colleagues from Delphix, especially Dan Kimmel, who started the initial prototype with me, and Matt Ahrens for guiding me every step of the way. I’d also like to thank Alexander Motin for porting the feature to FreeBSD, and Marius Zaborski for promoting it. •

Serapheim Dimitropoulos is a software engineer working on ZFS at Delphix. His main contributions to the project are the Log Spacemap and Pool Checkpoint features. When not programming, Serapheim spends his time playing soccer, salsa dancing, and playing classical guitar.

## REFERENCES

Serapheim's *ZPool Checkpoint* tutorial. <http://sdimitro.github.io/post/zpool-checkpoint/>

Marius's *ZPool Checkpoint* tutorial. <http://oshogbo.vexillum.org/blog/46/>

*Implementation Overview Block Comment* in the source. [https://github.com/freebsd/freebsd/blob/master/sys/cddl/contrib/opensolaris/uts/common/fs/zfs/spa\\_checkpoint.c](https://github.com/freebsd/freebsd/blob/master/sys/cddl/contrib/opensolaris/uts/common/fs/zfs/spa_checkpoint.c)

# FreeBSD 12.0 Toolchain Update

By John **Baldwin** & Ed **Maste**

FreeBSD 12.0 continues the trend in recent FreeBSD releases of transitioning away from obsolete GPLv2-licensed toolchain components to modern ones. During the 12.0 development cycle this work was primarily focused in two areas: using more of the LLVM-based toolchain where possible and improving support for a modern GNU toolchain. As a result of these changes, developers began to use features exclusive to modern toolchains near the end of the 12.0 development cycle.

## Expanding LLVM Toolchain Use

FreeBSD 10.0 and 11.0 used the clang C and C++ compiler as the default system compiler for the x86 and little-endian ARM architectures. However, the object files generated by clang were linked into binaries and executables by the GNU BFD linker. For x86 and 32-bit ARM, the legacy GPLv2 linker in the base system was used. For 64-bit ARM, a GPLv3 linker had to be installed from ports.

Over the past few years the LLVM developers have made substantial improvements to the LLD linker. FreeBSD developers have contributed to this effort both with patches and also by using FreeBSD's base system and ports as a large testing base to flesh out bugs and missing features. As a result of this work, FreeBSD 12.0 now ships LLD as the linker for 64-bit x86, 64-bit ARM, and ARMv7 architectures replacing the use of the GNU BFD linker. 32-bit x86 systems also use LLD as the linker for compiling the base system and kernel. (For 32-bit x86 a small number of ports rely on default options or behavior specific to GNU ld, and it is still installed as `/usr/bin/ld`.)

In addition to LLD changes, support for other architectures has improved in LLVM. Support for both MIPS and PowerPC has matured in LLVM. Some of these fixes have been submitted by FreeBSD developers while others have come from

other members of the LLVM community. While these architectures are not yet ready to use an LLVM-based toolchain in FreeBSD 12.0, progress is being made. For example, 64-bit MIPS should be able to use both clang and LLD from LLVM 7.0 once that is merged.

## External GNU Toolchain

Architectures not currently supported by the LLVM toolchain also need to transition to a more modern toolchain. Newer architectures such as RISC-V are not supported by the GPLv2 toolchain in the FreeBSD tree. In addition, building the base system with a modern GNU toolchain for architectures supported by LLVM provides users with a choice in toolchains. Rather than maintaining a GPLv3-licensed toolchain in the base source tree, modern GNU toolchains are built as separate packages using the ports framework.

GNU toolchain packages come in two varieties. The first set of packages installs GCC and binutils as an additional toolchain in `/usr/local` and can be used for either native or cross builds. The second set of packages builds a base system compiler that installs GCC and binutils into `/usr` as the default toolchain.

The additional toolchain packages consist of three separate packages for each architecture: `arch-binutils`, `arch-gcc`, and `arch-xtoolchain-gcc`. The last package depends on the other two packages, and all of these packages are built from ports in the devel category. Once an external toolchain is installed, it can be used to build kernels and the base system via the `CROSS_TOOLCHAIN` make variable. The value passed to `CROSS_TOOLCHAIN` is `"arch-gcc"`. For example, to build a 32-bit MIPS world, one would perform the steps in the following example.

## Example: Building 32-bit MIPS World with External GCC:

```
# pkg install mips-xtoolchain-gcc
# cd /path/to/src
# make buildworld TARGET_ARCH=mips CROSS_TOOLCHAIN=mips-gcc
```

The base system packages consist of two packages: `freebsd-binutils` and `freebsd-gcc`. These packages are built from the `base/binutils` and `base/gcc` ports. Unlike the additional toolchain packages, these packages replace components in the base system toolchain such as `/usr/bin/cc` and `/usr/bin/ld`. The ports for these packages (along with `pkg(8)` itself) can be cross-built from a non-native host. This will permit the Project to provide toolchain packages even on architectures for which the Project does not provide full package repositories.

Even when using a GNU toolchain, many toolchain components are still provided from other sources. For example, all FreeBSD architectures with a modern toolchain use `libc++` from LLVM as the C++ runtime library. Utilities such as `strip(8)` and `objcopy(8)` are provided by the ELF Tool Chain project.

FreeBSD 11 included support for additional toolchain packages and `CROSS_TOOLCHAIN`. During the FreeBSD 12 development cycle, work has focused on further refining this support. For example, the support for the `--sysroot` flag has been improved by both patches and configuration changes to the toolchain packages. In addition, the build system was updated to be more friendly to external toolchains with changes such as using the compiler driver to link binaries whenever possible and supporting different MIPS ABIs such as N32.

The base system toolchain packages have also been under active development over the past two years. Support has been added for the MIPS and x86 architectures. The same fixes for `--sysroot` support applicable to the additional toolchain packages also fixed similar issues with the base system packages. While they are not yet in a state to replace the legacy GPLv2 toolchain for any architectures in FreeBSD 12.0, developers have been able to build and boot a self-hosted world and kernel on 32-bit MIPS.

## Using Modern Toolchain Features

One of the benefits of moving to modern toolchains is the ability to use new toolchain features in the base system. Much of the work on toolchains prior to FreeBSD 12 focused on bringing on supporting a permissively-licensed toolchain on x86 architectures as well as supporting new architectures such as 64-bit ARM. However, FreeBSD was still treating the legacy GPLv2 toolchain as the lowest-common-denominator for deciding which toolchain features the base system used.

Toward the end of the FreeBSD 12 development cycle this focus has shifted. As LLD has matured, FreeBSD has achieved the goal of a permissively-licensed toolchain on the ARM and x86 architectures.

As a result, developers have now begun to focus on using (and in some cases requiring) features only supported by modern toolchains.

A prominent example of this is the use of indirect functions on x86 kernels. Indirect functions are a toolchain feature that permit a linker to invoke a function when resolving a symbol to determine what address the symbol should resolve to. This is commonly used to provide routines optimized for different processors. For example, a C runtime library might provide versions of string functions that use AVX or SSE instructions and choose the optimal version for the current CPU. FreeBSD 12.0 kernels for x86 architectures make use of this feature to provide optimized routines for memory copies, TLB flushes, and FPU state management. FreeBSD amd64 kernels also use indirect functions to support the Supervisor Mode Access Prevention (SMAP) security feature. Looking forward, the FreeBSD 13 development branch has already begun using indirect functions in the userland C runtime library to provide optimized routines for memory clearing and memory copies. The use of indirect functions will continue to expand in the future in both userland and the kernel.

## Conclusion

FreeBSD 12.0 marks another milestone in toolchain development. The ARM and x86 architectures now use modern, permissively licensed compilers and linkers. Support for external GCC toolchains is maturing. FreeBSD 13 will no longer use GPLv2 bits in the base system toolchain on any architectures. As a result, FreeBSD developers will accelerate the adoption of new toolchain features in the future. This will range from expanding the use of indirect functions, to enabling new features such as link-time optimization (LTO), build identifiers, compressed debug information, and more. ●

**JOHN BALDWIN** is a systems software developer. He has directly committed changes to the FreeBSD operating system for 19 years across various parts of the kernel (including x86 platform support, SMP, various device drivers, and the virtual memory subsystem) and userspace programs. In addition to writing code, John has served on the FreeBSD core and release engineering teams. He has also contributed to the GDB debugger and LLVM. John lives in Concord, California, with his wife, Kimberly, and three children: Janelle, Evan, and Bella.

**ED MASTE** manages project development for the FreeBSD Foundation and works in an engineering support role with the University of Cambridge Computer Laboratory. He is also a member of the elected FreeBSD core team. Aside from FreeBSD and LLVM projects, he is a contributor to several other open-source projects. He lives in Kitchener, Canada, with his wife, Anna, and sons, Pieter and Daniel.



# WeGetletters by Michael W Lucas

Dear ed(1),  
I keep hearing about FreeBSD conferences.  
Should I go?

Sincerely  
Random Sysadmin

Dear Random,  
Yes, you should go. Go, now. Get out of here  
and go!

(time passes)

You're still here?

Sigh. Fine. Let's talk about conferences.

You'll find FreeBSD content at all sorts of technical conferences. Even most Linux events have one or two FreeBSD talks. Ohio LinuxFest has so much, they might as well declare an official FreeBSD track. You'll find FreeBSD content at commercial conferences like Usenix and at a whole slew of security conferences.

But really, you want to hit a dedicated BSD event: BSDCan, AsiaBSDCon, or EuroBSDCon. The smaller, less regular conferences like NYCBSDCon, MeetBSD, vBSDCon are all pretty cool as well. If you speak Portuguese there's Brazil's BSDDay.

BSDCan is at the University of Ottawa, Canada, usually in June. AsiaBSDCon is at the University of Tokyo, Japan, in March. EuroBSDCon, in September, moves each year—not only to a different city, but a different country. I guess Europeans are easily bored? All of these conferences have massive amounts of excellent FreeBSD content. You'll also find all sorts of stuff from other BSDs as well. You can binge on FreeBSD. Drown in it. You can even spread FreeBSD on the floor of your hotel room and roll around in it but do have a shower afterwards.

While the talks are overwhelmingly excellent (except when they give that Lucas guy a platform) the part I most appreciate is the hallway track. When you get together all these really smart programmers, sysadmins, users, and hangers-on, it's like spending a weekend inside the Pinball Machine of Knowledge. Show up to a BSD conference with a problem? You can find people who have solved your problem, or some-

thing much like it. Come looking for an interesting problem to spend some time on? There's a whole bunch of people with problems, and some of them you can solve. And these folks keep up on technology. You'll get offered current ideas, along with older ideas that still work really well—much like FreeBSD itself.

Each time I go to a BSD conference, I leave with more ideas than I can fail to deploy in the next year.

Conferences also give you a chance to interact directly with a bunch of FreeBSD developers. You shouldn't barge up to every developer in the place with your pet annoyance, but it's really easy to ask a question over lunch and see what happens. People code FreeBSD either to a specification (for Foundation-sponsored work) or to solve their own problems.

A developer might be able to point you at a solution, suggest a different tool, or give you some pointers on developing your own solution and contributing it back to the community. Don't say you can't become a committer—at one point, every single committer was just as clueless as you.

These conferences are important enough to the FreeBSD community that the FreeBSD Foundation sponsors them. A two-day FreeBSD developer summit precedes each conference. Committers show up from all over the world to discuss FreeBSD improvements, works in progress, and what comes next. Sometimes five minutes with a whiteboard can cut out months of emails. By subscribing to this magazine, you're supporting those meetings.

There's a last reason for going to a BSD conference, though:

They're a whole lot of fun.

The fun is built into the programming.

Food is fun. EuroBSDCon takes advantage of the locale to host spectacular dinners. You haven't lived until you've eaten spectacular food in a Parisian catacomb or on the only sand beach on Malta. Everyone tells me Japan is magnificent, what with the geeky shopping and amazing seafood. And poutine is sufficient reason to visit Canada. All of these conferences are held in places that have a bunch to interest we geeky sorts, from museums to mints. Bring the family.

They can go out and do cool stuff while you nerd out.

If all that doesn't sell you on attending a conference, I can't help you.

Dear ed(1),  
Okay, I want to go. But I can't afford it.  
Any suggestions?

—Random Sysadmin

RS,  
BSD conferences are decidedly noncommercial. They're designed to be as inexpensive as possible. When you attend BSDCan, you get access to student housing. But maybe you're flat-out broke, can't prevail upon your hypothetical employer to foot the bill, or haven't figured out how to save part of your salary for the things you want. (If it's that last one; fix that.)

There are two ways to get to a conference on the cheap.

The simplest is to present. Unlike many commercial conferences, BSDCan, AsiaBSDCon, and EuroBSDCon pay travel and housing expenses for speakers. As a BSDCan committee member, I know that we've paid airfare for speakers flying in from South Africa and Australia. It's not first class, either the airfare or the accommodations, but you'll only go back to your hotel room to collapse in exhaustion, so who cares? Your presentation doesn't have to be highly technical; conferences want a mix of programming, systems administration, and war stories.

The other method? FreeBSD contributors can approach the FreeBSD Foundation for a travel grant. You'll have to fill out paperwork explaining your need and how your presence will enhance FreeBSD, but it's not unspeakably onerous. I've seen a surprising number of folks attend conferences on the Foundation's dime.

Why would the Foundation fly people in?

Bringing talented folks to a FreeBSD devsummit, letting them meet people and learn about existing issues they could help with, improves the chances that they'll pitch in to help FreeBSD.

Pick a conference on your continent and go!

Dear ed(1),  
There's no conference on my continent.

—RS

How did I let the Journal Editorial Board talk me into answering these letters?

FreeBSD is created by volunteers. FreeBSD conferences are the same. If no conference exists on your continent, then create one. You don't need anyone's permission. There's no central arbiter of who may run FreeBSD cons. All you need is space to hold the con, a few folks to help organize it, and Internet-based publicity.


If possible, attend one of the existing conferences so you can see how they run. Watch Dan Langille's video on running a BSD conference. Shamelessly steal his advice.

Conferences like MelbourneBSDCon or BSDNigeria will only help expose more people to FreeBSD.

And that's how our community grows. One continent, one conference, one conversation at a time. ●

---

Michael W Lucas (<https://mwl.io>) is the author of too many books, including *Absolute FreeBSD*, *FreeBSD Mastery: Specialty Filesystems*, and *git commit murder*. Send your questions to [letters@freebsdjournal.com](mailto:letters@freebsdjournal.com). Letters will be answered in the order in which they amuse, annoy, or inspire the columnist, and may be edited for his own purposes.



# Write For Us!

Contact Jim Maurer  
([jmaurer@freebsdjournal.com](mailto:jmaurer@freebsdjournal.com))  
with your article ideas.

# conference **REPORT**

by Roller Angel

## EuroBSDcon 2018—Bucharest, Romania



### PRIOR TO THE CONFERENCE

September found me in Bucharest for EuroBSDcon 2018, and my first talk at a BSD conference on the other side of the world! Once my proposal—about teaching, learning, and building confidence, all focused around BSD technology—was accepted, I had immediately started making arrangements for what felt like a once-in-a-lifetime opportunity, but what I know now was a door opening.

### WEDNESDAY

I arrived early Wednesday Romanian time, which was still late-Tuesday Colorado time. Ironically, the name of the hotel where speakers were housed was quite fitting—Hotel Yesterday! Since it was early in the day, I wandered around the streets of Bucharest sightseeing while the hotel prepared my room for check-in. The city has a deep, old history with a population of about 2.1 million; New York City has 8.6 million. But the population density is about the same.

### THURSDAY

I walked from the hotel to the conference, which was held at Politehnica University of Bucharest, the largest technical university in Romania, founded in 1864 as the School of Bridges and Roads, Mines and Architecture. It was renamed in 1920. A conference attendee named Pedro was the first person I met. We registered at the same time and then sat together. We immediately started talking about our FreeBSD laptops and shared various tips and tricks. For instance, Pedro demonstrated how easy it is to tether the Internet via USB from Android to

FreeBSD: Enable USB tether, plug in the cable, run `dhclient` followed by the name of the device (`ue0` or similar), wait a few seconds and the Internet from the conference WiFi is connected on FreeBSD through USB tether on Android. Pedro was running FreeBSD-CURRENT on his laptop, and I run FreeBSD-RELEASE on mine. When Benedict Reuschling joined us, we walked together to his Ansible tutorial.

Reuschling's tutorial was full of useful examples and included in-depth explanations of the various parts of the system while demonstrating how to use them in practical ways. I really enjoyed learning so many techniques for managing servers.

Next, I attended the Poudriere tutorial by Niclas Zeising, who did an amazing job introducing Poudriere and demonstrating some functions one might want to use when compiling ports. He also walked participants through the steps of creating a port. Tutorial participants were invited to attend the Devsummit dinner, so I decided to go along and ended up at a table with Deb Goodkin and Benedict Reuschling. Shortly after the dinner, Mahdi Mokhi joined the table and we discussed what I have been working on with FreeBSD and Python.

### FRIDAY

Pedro was there again, and this time we talked about setting up email for my BSD.pw domain. He suggested caesonia mail server, which uses OpenBSD and is made by the same people who make the vedetta firewall. Then it was time for Friday's tutorial, an all-day session focused on BGP and taught by the brilliant Peter Hessler. It was the culmination of several years of feedback and iteration, so I was able to experience an advanced version of the tutorial that included a web interface with two terminals side-by-side, all ready to go. Instead of having each tutorial participant install two OpenBSD virtual machines configured to run BGP, Peter hosted two OpenBSD virtual machines for each student on his laptop that was connected to the EuroBSDcon WiFi where he set up the URL `bad.network`. Each participant received a slip of paper that included the URL, username, and password needed to access a particular pair of machines using just a web browser. This worked incredibly well as the entire room was able to get up and running quickly. The tutorial included

information on networks, subnets, and static routes, which are a prerequisite to understanding BGP and its important role on the Internet. BGP stands for Border Gateway Protocol and it was designed to allow AS routers—known as autonomous systems—to share routing tables. It basically allows computers on the Internet to find each other in order to connect. I had a great time learning about BGP, sharing routing tables with other participants and learning about the importance of filtering the routes announced by peers. Hessler asked participants to share routes and then demonstrated what can happen when you don't use proper filtering. A peer can just announce every address—yes, every single address—and make the processing on your machine go crazy. Everyone was pleasantly entertained and had a great time. I highly recommend this tutorial to anyone wanting to learn about BGP.

After the all-day tutorial, I headed over to the Devsummit dinner. I sat next to Marshall Kirk McKusick and ended up talking to him all night. In fact, his husband actually told me at the end of the night that I had monopolized Kirk, and so I was responsible for making sure he made it back to the hotel safely. Oops! I guess I deserved that. I couldn't help it! Sitting next to this revolutionary FreeBSD committer was my opportunity to ask a lot of questions. What an experience that was.

## SATURDAY

The morning announcements were given by Michai Stanek and then he introduced his colleague, Costin Raiciu, from the Politehnica University of Bucharest. Costin gave the morning keynote on "Lightweight Virtualization." He presented some interesting research on the topic and an example of a very tiny Python environment running on one of the machines.

Next, "Hacking Together a FreeBSD Presentation Streaming Box for as Little as Possible," by Tom Jones, provided some useful examples on how one might record and stream BSD conference slides along with the speaker's voice. The conference was designed to provide a lot of varied information in a short time, and so I was then on to Kristaps Dz's "OpenBSD and Diving." Not only humorous and full of great pictures, this presentation was also full of great facts on the use of OpenBSD for managing and editing media. All the media transfer, editing, and color correcting was done on OpenBSD. Kristaps asked if anyone had suggestions for good media management applications, and I recommended Plex. I told him I'd only seen it available in the FreeBSD ports collection, but he assured me that was not a

problem and he would look into porting it to see if it fits his workflow. I think it will, as I use it every day and it does a good job of removing a lot of the complexity associated with media management as well as providing a stable interface for working with media.

In the talk by Ingo Schwarze on some of the headaches related to working with Markdown, he asked for help with a CSS issue in the main `mandoc.css` file that controls the presentation of man pages on the web for OpenBSD. I'm familiar with CSS and offered to take a look at the issue to see if I could help. The moment I saw the code, I recognized the issue and knew what it would take to make the fix. I just needed to find the time to sit down and write the code. Ingo assured me that I could take my time and contact him at my leisure. Not only are there great opportunities to take in information, but also opportunities to be a part of the public conversation.

The next talk was "What's in Store for NetBSD 9.0," in which Sevan Janiyan shared the milestones reached in the NetBSD project, including details about what will be in the next release. I highly recommend checking out his slides. All conference slides are available for download via the [EuroBSDCon.org](http://EuroBSDCon.org) website. The second keynote, "Some Computing and Networking Historical Perspectives," by Ron Broersma, was a real treat for all in the room. Ron brought along various historical artifacts from his early career, including his work on Arpanet. The entire auditorium loved the talk and you could feel the interest growing in the room each minute. Ron would periodically pull out an artifact from under the podium—for example, an old boot card that used to come with the big, million-dollar computers in use at the time. You actually needed to use the card to know where in memory to set the program to be able to boot the thing. You had to manually type in the codes to get everything lined up so that boot would function properly! After the talk, Ron welcomed people to come up and look at the items he had brought with him.

## SUNDAY

On the last day of the conference, I was running a little late but hurried over to the venue in time to catch "Pledge and Unveil in OpenBSD," by Bob Beck. Bob understands the guts of operating system development and presented some best practices and various ways to write safe programs on OpenBSD with pledge and unveil.

I was totally lost at "Integrate libFuzzer with the NetBSD Userland," a very advanced and technical talk by Yang Zheng. Perhaps one day I will get to

the point where I can grok this topic.

"DeforaOS, NetBSD, Future Internet," by Pierre Pronchery, was a very intellectual talk. Pierre is a very thoughtful developer who pushes the limits and has a fantastic understanding of the underlying technology. I learned a lot about why things are the way they are and was given a challenge to make things better.

"Debugging Lessons Learned as a Newbie Fixing NetBSD," by Maya Rashish, followed a newbie through the process of contributing to NetBSD and detailed some of the things a person new to the Project should be aware of. After Maya's talk, I caught the tail end of "FreeBSD Graphics," by Niclas Zeising, where I asked a question about a hiccup in the process of compiling `drm-stable-kmod`. There is an ignore flag in the Makefile that needs to be removed if one wants to compile the port on FreeBSD 11.2-RELEASE, which is what I'm using. Niclas said it must have been an oversight and that it should work fine on 11.2. I let him know I removed the ignore flag from the Makefile and compiled `drm-stable-kmod` on my 11.2-RELEASE laptop and it has been working like a charm.

I was up next with my talk, "Being a BSD User." Minutes before I started, the adrenaline kicked in.

Despite my nervous reaction to public speaking, the talk went quite well. The audience was engaged and asked questions. I shared information on my experiences with BSD technologies and how teaching young scientists about the technology is a great experience. Learning about all the cool things you can do with the BSDs and about the amazing community is an experience I enjoy bringing to others. I had some great feedback from Allan Jude, Kirk McKusick, and others. Although I was initially a little concerned about the 45-minute block I needed to fill, once I began talking, my passion took over and I was able to fill the time slot with no problem.

There were 181 attendees from 37 countries. Groff was brought to the conference by Sevan Janiyan and put in the care of Deb Goodkin at the closing session. •

---

Roller Angel is an avid BSD user who enjoys all the amazing things that can be done with BSD technology. He has taught programming workshops based on FreeBSD and is working on building an online training platform for teaching BSD and related technologies. See [BSD.pw](http://BSD.pw) for more information.

---

## ZFS experts make their servers **ZING**

Now you can too. Get a copy of.....

**Choose ebook, print, or combo. You'll learn to:**

- Use boot environment, make the riskiest sysadmin tasks boring.
- Delegate filesystem privileges to users.
- Containerize ZFS datasets with jails.
- Quickly and efficiently replicate data between machines.
- Split layers off of mirrors.
- Optimize ZFS block storage.
- Handle large storage arrays.
- Select caching strategies to improve performance.
- Manage next-generation storage hardware.
- Identify and remove bottlenecks.
- Build screaming fast database storage.

WHETHER YOU MANAGE A SINGLE SMALL SERVER OR INTERNATIONAL DATACENTERS, SIMPLIFY YOUR STORAGE WITH **FREEBSD MASTERY: ADVANCED ZFS**.

GET IT TODAY! **Link to:**

**<http://zfsbook.com>**





# FreeBSD<sup>®</sup> JOURNAL



# The FreeBSD Journal is going Free!

## Yep, that's right. Free.

Starting with the January/February 2019 issue, the voice of the FreeBSD Community and the BEST way to keep up with the latest releases and new developments in FreeBSD is now openly available to everyone.

**Don't miss a single issue!**

Find out more at: [freebsd.foundation/journal](https://freebsd.foundation/journal)

### 2019 Editorial Calendar:

- Getting Started with FreeBSD (Jan/Feb 2019)
- Debugging and Testing (March/April 2019)
- FreeBSD for Makers (May/June 2019)
- Containerization (July/Aug 2019)
- Security (Sept/Oct 2019)
- Network Virtualization (Nov/Dec 2019)

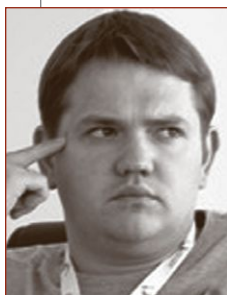
# new faces

## of FreeBSD

BY DRU LAVIGNE

This column aims to shine a spotlight on contributors who recently received their commit bit and to introduce them to the FreeBSD community. In this month's column, the spotlight is on **Sergey Kozlov**, who received his ports bit in September, and **Vinícius Zavam**, who received his ports bit in October.

Tell us a bit about yourself, your background, and your interests.



• **Sergey:** My name is Sergey Kozlov and I'm a FreeBSD addict. I was born and spent most of my life in the Ukraine, Kyiv, but since 2015, I've been living in Gdansk, Poland. I'm a Python programmer, sysadmin, and tester. I'm working at Intel on automatic testing of Intel's FreeBSD wired Ethernet drivers (em, igb, ix, ixl, etc.).



• **Vinícius:** I was born and raised in Fortaleza, with roots in Rio de Janeiro and Minas Gerais. My family's name comes originally from Italy and was misspelled during the Italian migration to Brazil, so I'm also the inheritor of the tasty tradition of dipping bread in coffee. The original family's name is Zavani. Now I'm living in Germany

(and "crazy" homesick, especially in winter) working for cleverbridge, which employs a massive use of FreeBSD.

As a kid, I got into computers after encountering a TK 85 and later graduated in computer engineering. I enjoy learning different languages. I'm engaged and motivated by the use and benefits of free and open-source software (FOSS), especially the ones under the BSD license, and have made contributions and donations to different projects over time. In the last few years I have been working to get more privacy-enhancing technologies (PET) support working on \*BSD operating systems. These efforts brought me to the TorBSD Diversity Project (TDP, <https://torbsd.org/2016/12/17/welcome-aboard-vinicius.html>) where I became a core team member (<https://www.torproject.org/about/corepeople.html.en#egypcio>).

How did you first learn about FreeBSD and what about FreeBSD interested you?

• **Sergey:** When I was a beginner sysadmin, I worked mostly on Windows Server, but wanted to learn UNIX-like operating systems, as I had heard so much about them. I started with the most popular, Linux, and I was trying to learn it for quite a while. I installed different distributions, versions, flavors, all the while looking on the Internet for books and articles related to Linux but couldn't learn a thing. I just felt that I hadn't started at the right place and didn't see a clear path for mastering my Linux skills. I was very surprised that all those operating systems were even called Linux, because I could hardly find anything common in them.

Later, I heard about FreeBSD from one of my friends. He said that somewhere out there, in the land of heavily loaded web servers and routers, there's a caste of old sysadmins with long beards who prefer tidiness and order. Those guys know everything about how their tools work and the OS they're using is called FreeBSD. And that's exactly what I was looking for! I downloaded the ISO and got a copy of Michael W Lucas's *Absolute FreeBSD* and that's where the world of UNIX opened to me. One thing at a time, step by step, I was getting more and more knowledge about what I was really doing and how the whole thing worked "under the hood." This feeling of order and clarity is what got me hooked on FreeBSD.

• **Vinícius:** My first steps into FreeBSD were about 15 years ago while I was working for an ISP in my hometown. The person who introduced me to FreeBSD was a good friend I met on the BRASnet Internet Relay Chat (IRC) network. Sadly, I was the only one at the ISP who really got into \*BSD and I was even running it on my workstations. After moving forward, I didn't abandon

Beastie and kept using FreeBSD for everything I could, even if others said I was just wasting my time—which I heard a lot.

It's still not easy for me to describe exactly what brought me to FreeBSD. I cannot just pick one single feature and say it was that. But there wasn't a single time that running it on my laptop or on any server got me mad or disappointed. The OS is rock solid, the Project and its documentation are well maintained, the developers care about the community and are in touch to get new features and/or hardware support working. And I like the license, the great companies that adopt FreeBSD, the research/products/solutions it empowers, the standards it follows, the certifications it's compliant to.

When FreeBSD landed on hardware like RaspberryPi and BeagleBone, I got very little sleep but had lots of fun. I got familiar with Poudriere and presented it together with a cross-compiling setup to the Institute of Technology (where I had graduated) during one of the Compilers 101 classes. At the very same Institute (Instituto Federal de Educação, Ciência e Tecnologia do Ceará—IFCE), I talked about KAME/IPv6 on FreeBSD and recommended the net/sysadmins use it to handle packet filtering. At the lab where I did my research (things like playing with Docker+Jails or developing an embedded system), I was also the net/sysadmin. Most of the servers were running FreeBSD, and one of the official FreeBSD Brazilian mirrors was running at that lab (it was the only one in Brazil serving the /snapshots folder, by the way).

#### How did you end up becoming a committer?

• **Sergey:** It all started when I found two pieces of software: p910nd and mjpg-streamer. I needed them, but I was forced to use Linux because they weren't available in the ports collection. This made me angry—"Why can they use it and we can't? How hard could it be to make such a small app work on FreeBSD?" This is when I decided to make my own port. I made several—every time someone at my workplace approached me and said, "Hey, we need <app\_name>, but there's no version for FreeBSD on the website," I would just smile and say, "Don't worry, I'll port it by tomorrow."

After a while I joined Intel where I was able to (finally!) make working on ports part of my job duties. Many Phabricator reviews later, sbruno@ apparently got tired of committing my code and decided that it was time for me to start breaking stuff myself.

• **Vinicius:** Well, time flies very fast! It seems like only yesterday when I ran sysinstall for the first time and upgraded my machines using cvsup, when I was subscribed to the FreeBSD Users Group Brazil mailing list. At that time, I was giving talks, facilitating tutorials, writing articles about FreeBSD, and getting people familiar with its features and what it's capable of. Sometime in between, I engaged in a great effort to translate the FreeBSD Handbook to Portuguese, and then I helped review and update a book chapter about network and Internet services powered by FreeBSD (FreeBSD: O Poder dos Servidores em Suas Mãos). One day in 2015, I found myself organizing the very first international BSD conference in Brazil!

Finally, a couple months ago I attended EuroBSDcon 2018, and there I met some other committers. During one of the coffee breaks and an off-the-record chat about PET support and alternatives to freebsd-update and what's going on regarding pkg-base and bhyve development, araujo@, beat@, and rene@ decided to punish me with a commit bit.

#### How has your experience been since joining the FreeBSD Project? Do you have any advice for readers who may be interested in also becoming a FreeBSD committer?

• **Sergey:** Before joining, I was eagerly searching for any news about the Project. I checked out every change on the wiki and looked forward to every quarterly status report. All the people working on the system were nothing less than rock stars for me. And then, all of a sudden, I get my commit bit and go to DevSummit 2018 in Bucharest. I'm sitting in the same room with all those stars, discussing the future of the Project and being treated like one of their own. I just couldn't believe it. In addition to all the excitement I had, lots of people wanted to help with my "new committer steps," like setting up mail, adding myself to docs, etc.; des@ even accidentally reset his password while helping me with SVN! It just really felt like a family.

For anyone who wants to become a committer, I can suggest three things: work, socialize, behave. First of all, make sure you're contributing to the Project on a regular basis. It may not be every day or week, but it has to be regular. Also, make sure to join one of the IRC channels and introduce yourself, start a conversation with someone. To be honest, I wasn't doing that, as I was too shy, and only after I joined the Project, did I realize how valuable

## new faces of FreeBSD

this can be. And last but not least, respect people around you, their work, time, and opinion. Make sure you fit into the family.

- **Vinícius:** I have gotten great input from many developers involved with the Project and learned more about FreeBSD internals over these last months—and that's amazing! There is really good stuff going on. In the past I just chatted, mostly with Brazilians, on IRC, and shared thoughts or experiences regarding all kinds of scenarios where FreeBSD is involved. Now, I can make a direct impact and contribute to solving other users' or even other developers' issues—especially when we all use it every single day on servers and worksta-

tions or embedded devices.

Before I officially got the title of ports committer, I was already maintaining a few ports and adopting others slowly. It is a great journey and a lot of fun!

What else can I say to others wanting to contribute? Just do your thing and have fun doing it. Contribute small changes, submit your patches, get stuff done. It's not a race, and it may take time. Chill. Help others and you will also help yourself.

.....  
**DRU LAVIGNE** is a FreeBSD doc committer and the author of *BSD Hacks* and *The Best of FreeBSD Basics*.

# Thank you!

The FreeBSD Foundation would like to acknowledge the following companies for their continued support of the Project. Because of generous donations such as these we are able to continue moving the Project forward.



Are you a fan of FreeBSD? Help us give back to the Project and donate today! [freebsd.foundation.org/donate/](http://freebsd.foundation.org/donate/)

Please check out the full list of generous community investors at [freebsd.foundation.org/donors/](http://freebsd.foundation.org/donors/)

Iridium



Platinum

# NETFLIX

Gold

facebook

JUNIPER  
NETWORKS

Silver



VERISIGN



vmware



Microsoft

Tarsnap

“Congratulations to the RE team and all the contributors for the release of 12.0! We’d also like to thank our community of donors whose support helps us fund a full-time staff member dedicated to leading the Release Engineering team and overseeing the release process.”



JAN., FEB., & MARCH

BY DRU LAVIGNE

# Events Calendar

The following BSD-related conferences will take place during the first quarter of 2019.



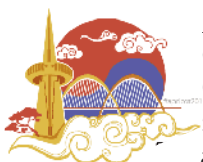
## **SANOG33 • Jan 9–16 • Thimphu, Bhutan**

<https://www.sanog.org/sanog33> • The 33rd Conference of South Asian Network Operators Group (SANOG) will provide a valuable opportunity to contribute to discussions about Internet operations, technologies and development.



## **FOSDEM • Feb 2 & 3 • Brussels, Belgium**

<http://fosdem.org/2019/> • FOSDEM is one of the largest annual European open-source events and is free to attend. There will be a BSD developer room and a FreeBSD table in the expo area. The FreeBSD Foundation is sponsoring a one-day developer summit the day before the conference at the NH Brussels Carrefour de l'Europe hotel.



## **APRICOT 2019 • Feb 18–28 • Yuseong-gu, Daejeon South Korea**

<https://2019.apricot.net> • Representing Asia Pacific's largest international Internet conference, Asia Pacific Regional Internet Conference on Operational Technologies (APRICOT) draws many of the world's best Internet engineers, operators, researchers, service providers, users and policy communities from over 50 countries to teach, present, and do their own human networking.



## **SCALE • March 7–10 • Pasadena, CA**

<http://www.socallinuxexpo.org/scale/17x> • The 17th annual Southern California Linux Expo is the largest community-run open-source and free software conference in North America. There will be a FreeBSD booth in the expo area. There is a minimal cost to attend this conference.



## **FOSSASIA 2019 • March 14–17 • Singapore, Singapore**

<http://2019.fossasia.org/> • The FOSSASIA Summit, Asia's leading open technology conference for developers, startups, and IT professionals will take place at the Lifelong Learning Institute, Singapore.

## **ASIABSDCon • March 21–24 • Tokyo, Japan**



<https://2019.asiabsdcon.org/> • This annual conference is for anyone developing, deploying, and using systems based on FreeBSD, NetBSD, OpenBSD, DragonFlyBSD, Darwin, and MacOS X. AsiaBSDCon is a technical conference and aims to collect the best technical papers and presentations available to ensure that the latest developments in our open-source community are shared with the widest possible audience.

# svn UPDATE

by Steven Kreuzer

Very recently something exciting happened to me. One night I ran svn up in /usr/src, kicked off a build, and went to bed. When I woke up in the morning, I found my laptop sitting at a prompt asking me to log in to my FreeBSD 13.0 workstation. That can only mean one thing. We are currently in the middle of the FreeBSD 12.0 release cycle! For a while there was not much action in HEAD, as the release engineering team had frozen the tree to create a branch of what will become 12-STABLE. That freeze has been lifted and we once again are seeing lots of new features and exciting changes. Can you think of a better way to end the year?

**Add macros for reading performance counter CSRs on RISC-V—** <https://svnweb.freebsd.org/changeset/base/340399>

The RISC-V spec defines several performance counter CSRs such as: cycle, time, instret, hpm-counter(3...31). They are defined to be 64-bits wide on all RISC-V architectures. On RV64 and RV128 they can be read from a single CSR. On RV32, additional CSRs (given the suffix "h") are present that contain the upper 32 bits of these counters and must be read as well. (See section 2.8 in the User ISA Spec for full details.)

This change adds macros for reading these values safely on any RISC-V ISA length. Obviously, we aren't supporting anything other than RV64 at the moment, but this ensures we won't need to change how we read these values if we ever do.

**Implement get\_cyclecount(9) for RISC-V—** <https://svnweb.freebsd.org/changeset/base/340400>

Add the missing implementation for get\_cyclecount(9) on RISC-V by reading the cycle CSR.

**Enable non-executable stacks by default for RISC-V—** <https://svnweb.freebsd.org/changeset/base/340231>

**Enable use of a global shared page for RISC-V—** <https://svnweb.freebsd.org/changeset/base/340228>

machine/vmparam.h already defines the SHARED-PAGE constant. This change just enables it for ELF executables. The only use of the shared page currently is to hold the signal trampoline.

**Add new rc keywords: enable, disable, delete—** <https://svnweb.freebsd.org/changeset/base/339971>

This adds new keywords to rc/service to enable/disable a service's rc.conf(5) variable and "delete" to remove the variable.

When the "service\_delete\_empty" variable in rc.conf(5) is set to "YES" (default is "NO"), an rc.conf.d file (in /etc/ or /usr/local/etc) is deleted if empty after modification using "service \$foo delete".

**libcasper: introduce cap\_fileargs service—** <https://svnweb.freebsd.org/changeset/base/340373>

cap\_fileargs is a Casper service that helps to sandbox applications that need access to the filesystem namespace. The main purpose of the service is to make it easy to capsicumize applications that works on multiple files passed in argv.

**Sandbox head using capsicum—** <https://svnweb.freebsd.org/changeset/base/340376>

**Sandbox wc using capsicum—** <https://svnweb.freebsd.org/changeset/base/340374>

**Add load balancer program for netmap—** <https://svnweb.freebsd.org/changeset/base/340279>

Add the lb program, which is able to load-balance input traffic received from a netmap port over M groups, with N netmap pipes in each group. Each received packet is forwarded to one of the pipes chosen from each group (using an L3/L4 connection-consistent hash function).

**Allow configuration of several ipsec interfaces with the same tunnel endpoints —** <https://svnweb.freebsd.org/changeset/base/340477>

This can be used to configure several IPsec tunnels between two hosts with different security associations.

**Add support for non-ACPI battery method batteries—** <https://svnweb.freebsd.org/changeset/base/340832>

Remove the requirement that a device be an ACPI method battery to be supported as a battery. Require now that the device be in the battery devclass and implement the `get_status` and `get_info` functions. This allows batteries that are not ACPI method batteries to be supported.

**Permit local kernel modules to be built as part of a kernel build—** <https://svnweb.freebsd.org/changeset/base/339901>

Add support for "local" modules. By default, these modules are located in `LOCALBASE/sys/modules` (where `LOCALBASE` defaults to `/usr/local`). Individual modules can be built along with a kernel by defining `LOCAL_MODULES` to the list of modules. Each is assumed to be a subdirectory containing a valid Makefile. If `LOCAL_MODULES` is not specified, all of the modules present in `LOCALBASE/sys/modules` are built

and installed along with the kernel.

This means that a port that installs a kernel module can choose to install its source along with a suitable Makefile to `/usr/local/sys/modules/<foo>`. Future kernel builds will then include that kernel module using the kernel configuration's `opt_*.h` headers and install it into `/boot/kernel` along with other kernel-specific modules.

**Add minimal support for active AUX port multiplexers—** <https://svnweb.freebsd.org/changeset/base/340913>

Active PS/2 multiplexing is a method for attaching up to four PS/2 pointing devices to a computer. Enabling of multiplexed mode allows commands to be directed to individual devices using routing prefixes.

Multiplexed mode reports input with each byte tagged to identify its source. This method differs from one currently supported by `psm(4)` where so-called guest device (trackpoint) is attached to special interface located on the host device (touchpad) and later performs guest protocol conversion to special encapsulation packet format.

**STEVEN KREUZER is a FreeBSD Developer and Unix Systems Administrator with an interest in retro-computing and air-cooled Volkswagens. He lives in Queens, New York, with his wife, daughter, and dog.**



#### The FreeBSD Project is looking for

- Programmers • Testers
- Researchers • Tech writers
- Anyone who wants to get involved

#### Find out more by

**Checking out our website**  
[freebsd.org/projects/newbies.html](https://freebsd.org/projects/newbies.html)

**Downloading the Software**  
[freebsd.org/where.html](https://freebsd.org/where.html)

We're a welcoming community looking for people like you to help continue developing this robust operating system. Join us!

#### Already involved?

Don't forget to check out the latest grant opportunities at  
[freebsd.foundation.org](https://freebsd.foundation.org)

## Help Create the Future. Join the FreeBSD Project!

FreeBSD is internationally recognized as an innovative leader in providing a high-performance, secure, and stable operating system.

Not only is FreeBSD easy to install, but it runs a huge number of applications, offers powerful solutions, and cutting edge features. The best part? It's FREE of charge and comes with full source code.

Did you know that working with a mature, open source project is an excellent way to gain new skills, network with other professionals, and differentiate yourself in a competitive job market? Don't miss this opportunity to work with a diverse and committed community bringing about a better world powered by FreeBSD.

The FreeBSD Community is proudly supported by

